NASA/CR-1999-208978
ICASE Report No. 99-2

Object-oriented Design for Sparse Direct Solvers

Florin Dobrian
Old Dominion University, Norfolk, Virginia

Gary Kumfert and Alex Pothen
Old Dominion University, Norfolk, Virginia
and
ICASE, Hampton, Virginia

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA

Operated by Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NAS1-97046

January 1999

OBJECT-ORIENTED DESIGN FOR SPARSE DIRECT SOLVERS*

FLORIN DOBRIAN†, GARY KUMFERT‡, AND ALEX POTHEN‡

Abstract. We discuss the object-oriented design of a software package for solving sparse, symmetric systems of equations (positive definite and indefinite) by direct methods. At the highest layers, we decouple data structure classes from algorithmic classes for flexibility. We describe the important structural and algorithmic classes in our design, and discuss the trade-offs we made for high performance. The kernels at the lower layers were optimized by hand. Our results show no performance loss from our object-oriented design, while providing flexibility, ease of use, and extensibility over solvers using procedural design.

1. Introduction. The problem of solving linear systems of equations $\boldsymbol{Ax} = \boldsymbol{b}$, where the coefficient matrix is sparse and symmetric, represents the core of many scientific, engineering and financial applications. In our research, we investigate algorithmic aspects of high performance direct solvers for sparse symmetric systems, focusing on parallel and out-of-core computations. Since we are interested in quickly prototyping our ideas and testing them, we decided to build a software package for such experimentation. High performance is a major design goal, in addition to requiring our software to be highly flexible and easy to use.

Sparse direct solvers use sophisticated data structures and algorithms; at the same time, most software packages using direct solutions for sparse systems were written in Fortran 77. These programs are difficult to understand and difficult to use, modify, and extend due to several reasons: First, the lack of abstract data types and encapsulation leads to global data structures scattered among software components, causing tight coupling and poor cohesion. Second, the lack of abstract data types and dynamic memory allocation leads to function calls with long argument lists, many arguments having no relevance in the context of the corresponding function calls. In addition, some memory may be wasted because all allocations are static.

We have implemented a sparse direct solver using different programming languages at different layers. We have reaped the benefits of object-oriented design (OOD) and the support that C++ provides for OOD, at the highest layer, and the speed of Fortran 77 at the lower levels. The resulting code is more maintainable, usable, and extensible but suffers no performance penalty over a native Fortran 77 code. To the best of our knowledge, this work represents the first object-oriented design of a sparse direct solver.

We chose C++ as a programming language since it has full support for object-oriented design, yet it does not enforce it. The flexibility of C++ allows a software designer to choose the appropriate tools for each particular software component. Another candidate could have been Fortran 90, but it does not have inheritance and polymorphism. We need inheritance in several cases outlined later. We also wish to derive new classes for a parallel version of our code. We do not want to replicate data and behavior that is common to some classes. As for polymorphism, there are several situations when we declare just the interfaces in a base class and we want to let derived classes implement a proper behavior.

*This work was partially supported by the National Science Foundation grants CCR-9412698 and DMS-9807172, by the Department of Energy grant DE-FG05-94ER25216, and by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 while the third author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-2199.

†Department of Computer Science, Old Dominion University, Norfolk, VA 23529-0162. Email: dobrian@cs.odu.edu.

‡Department of Computer Science, Old Dominion University, Norfolk, VA 23529-0162, and ICASE, NASA Langley Research Center, Hampton VA 23681-2199. Email: {kumfert, pothen}@cs.odu.edu, {kumfert, pothen}@icase.edu. URL: www.cs.odu.edu/~pothen.

In this paper we present the design of our sequential solver. Work on a parallel version using the message-passing model is in progress. Object-oriented packages for iterative methods are described in [1, 2].

2. Overview of the problem. Graph theory provides useful tools for computing the solution of sparse systems. Corresponding to a symmetric matrix $\boldsymbol{A}$ is its undirected adjacency graph $G(\boldsymbol{A})$. Each vertex in the graph corresponds to a column (or row) in the matrix and each edge to a symmetric pair of off-diagonal nonzero entries.

The factorization of $\boldsymbol{A}$ can be modeled as the elimination of vertices in its adjacency graph. The factorization adds edges to $G(\boldsymbol{A})$, creating a new graph $G^+(\boldsymbol{A}, \boldsymbol{P})$, where $\boldsymbol{P}$ is a permutation that describes the order in which the columns of $\boldsymbol{A}$ are eliminated. Edges in G^+ not present in G are called fill edges and they correspond to fill elements, nonzero entries in the filled matrix $\boldsymbol{L}+\boldsymbol{D}+\boldsymbol{L}^T$ that are zero in $\boldsymbol{A}$.

The computation of the solution begins thus by looking for an ordering that reduces the fill. Several heuristic algorithms (variants of minimum degree or nested dissection) may be used during this step. The result is a permutation $\boldsymbol{P}$.

Next, an *elimination forest* $F(\boldsymbol{A}, \boldsymbol{P})$, a spanning forest of $G^+(\boldsymbol{A}, \boldsymbol{P})$, is computed. The elimination forest represents the dependencies in the computation, and is vital in organizing the factorization step. Even though it is a spanning forest of the filled graph, it can be computed directly from the graph of $\boldsymbol{A}$ and the permutation $\boldsymbol{P}$, without computing the filled graph. In practice, a compressed version of the elimination forest is employed. Vertices that share a common adjacency set in the filled graph are grouped together to form supernodes. Vertices in a supernode appear contiguously in the elimination forest, and hence a supernodal version of the elimination forest can be used.

The factorization step is split in two phases: *symbolic* and *numerical*. The first computes the nonzero structure of the factors and the second computes the numerical values. The symbolic factorization can be computed efficiently using the supernodal elimination forest. The multifrontal method for numerical factorization processes the elimination forest in postorder. Corresponding to each supernode are two dense matrices: a *frontal matrix* and an *update matrix*. Entries in the original matrix and updates from the children of a supernode are assembled into the frontal matrix of a supernode, and then partial dense factorization is performed on the frontal matrix to compute factor entries. The factored columns are written to the factor matrix, and the remaining columns constitute the update matrix that carries updates higher in the elimination forest.

Finally, the solution is computed by a sequence of triangular and diagonal solves. Additional solve steps with the computed factors (iterative refinement) may be used to reduce the error if it is large.

When the coefficient matrix is positive definite, there is no need to pivot during the factorization. For indefinite matrices, pivoting is required for stability. Hence the permutation computed by the ordering step is modified during the factorization.

Additional details about the graph model may be found in [3]; about the multifrontal method in [4]; and about indefinite factorizations in [5].

3. Design of the higher layers. At the higher layers of our software, the goal was to make the code easy to understand, use, modify and extend. Different users have different needs: Some wish to minimize the intellectual effort required to understand the package, others wish to have more control. Accordingly, there must be different amounts of information a user has to deal with, and different levels of functionality a user is exposed to.

At the highest level, a user is aware of only three entities: the coefficient matrix $\boldsymbol{A}$, the right hand side vector $\boldsymbol{b}$, and the unknown vector $\boldsymbol{x}$. Thus a user could call a solver as follows:

$$\boldsymbol{x} = \mathit{Compute}\,(\boldsymbol{A},\, \boldsymbol{b}),$$

expecting the solver to make the right choices. Of course it is difficult to achieve optimal results with such limited control, so a more experienced user would prefer to see more functionality. Such a user knows that the computation of the solution involves three main steps: (1) *ordering*, to preserve sparsity and thus to reduce work and storage requirements, (2) *factorization*, to decompose the reordered coefficient matrix into a product of factors from which the solution can be computed easily, and (3) *solve*, to compute the solution from the factors. This user would then like to perform something like this:

$$\begin{aligned} \boldsymbol{P} &= \mathit{Order}\,(\boldsymbol{A}), \\ (\boldsymbol{L},\, \boldsymbol{D},\, \boldsymbol{P}) &= \mathit{Factor}\,(\boldsymbol{A},\, \boldsymbol{P}), \\ \boldsymbol{x} &= \mathit{Solve}\,(\boldsymbol{L},\, \boldsymbol{D},\, \boldsymbol{P},\, \boldsymbol{b}). \end{aligned}$$

Here, $\boldsymbol{P}$ is a permutation matrix that trades sparsity for stability, $\boldsymbol{L}$ is a unit lower triangular or block unit lower triangular matrix, and $\boldsymbol{D}$ is a diagonal or block diagonal matrix.

At this level the user has enough control to experiment with different algorithms for each one of these steps. The user could choose a minimum degree or a nested dissection ordering, a left-looking or a multifrontal factorization. In addition, the user may choose to run some of the steps more than once to solve many related systems of equations, or for iterative refinement to reduce the error.

We organized the higher layers of our software as a collection of classes that belong to one inheritance tree. At the root of the tree we put the *Object* class, which handles errors and provides a debugging interface. Then, since the two basic software components are data structures and algorithms, and since decoupling them achieves flexibility, we derived a *DataStructure* class and an *Algorithm* class from *Object*. The first one handles general information about all structural objects and the second one deals with the execution of all algorithmic objects.

An important observation is necessary here. While full decoupling needs perfect encapsulation, the overhead introduced by some interfaces may be too high. Thus performance reasons forced us to weaken the encapsulation allowing more knowledge about several objects. For sparse matrices, for example, we store the data (indices and values) column-wise, in a set of arrays. We allow other objects to retrieve these arrays, making them aware of the internal representation of a sparse matrix. We protect the data from being corrupted by providing non-const access only to functions that need to change the data. Such a design implementation may be unacceptable for an object-oriented purist. However, a little discipline from the user in accessing such objects is not a high price for a significant gain in performance.

A user who does not want to go beyond the high level of functionality of the main steps required to compute the solution sees the following structural classes: *SparseSymmMatrix*, *Vector*, *Permutation* and *SparseLwTrMatrix*. The first class describes coefficient matrices, the second right hand side and solution vectors, the third permutations, and the fourth both triangular and diagonal factors. We decided to couple these last two because they are always accessed together and a tight coupling between them leads to higher performance without any significant loss in understanding the code. The derivation of these four classes from *DataStructure* is shown in Fig. 3.1.

At the same level the user also sees several algorithmic classes. First there are various ordering algorithms, such as *NestDissOrder* or *MultMinDegOrder*. Then there are factorization algorithms, like *PosDefLeftLookFactor*, *PosDefMultFrtFactor* or *IndefMultFrtFactor*. Finally, the solve step can be performed by

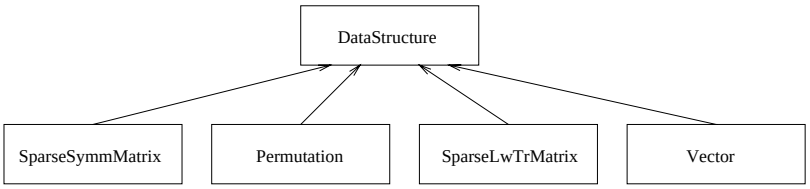

FIG. 3.1. *High level structural classes*

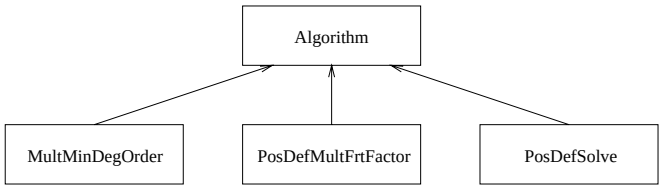

FIG. 3.2. *Some high level algorithmic classes*

PosDefSolve or *IndefSolve* algorithms. Figure 3.2 describes the derivation of some of these classes from *Algorithm*. Using them one can easily write a solver (positive definite, for concreteness) shown in Fig. 3.

More details are available beyond this level of functionality. The factorization is split in two phases: *symbolic* and *numerical*. The symbolic factorization is guided by an elimination forest. The multifrontal method for numerical factorization uses an update stack and several frontal and update matrices, which are dense and symmetric. Pivoting strategies for indefinite systems can be controlled at the level of frontal and update matrices during the numerical factorization phase. Figures 3.4 and 3.5 depict the derivation of the corresponding structural and algorithmic classes.

Classes such as *SparseSymmMatrix*, *SparseLwTrMatrix*, and *Permutation* are implemented with multiple arrays of differing sizes. Several of these are arrays of indices that index into the other arrays, so that the validity of the state of a class depends on not only the individual internal arrays, but the interaction between several of them.

In a conventional sparse solver, these arrays are global and some of them are declared in different modules. A coefficient matrix, a factor, a permutation, or an elimination forest is not a well defined entity but the sum of scattered data. This inhibits software maintenance because of the tight coupling between disparate compilational units.

There are also significant benefits in terms of type safety. For instance, a permutation is often represented as an array of integers. It could be that the index of the old number holds the new position or vice versa. We use *oldToNew* and *newToOld* to refer to the two arrays. The problem is that interpreting a *newToOld* permutation as an *oldToNew* permutation yields a valid operation, though an incorrect permutation. It is easy for users to reverse these two, particularly when the names "permutation" and "inverse permutation" are applied since there is no agreement on whether *newToOld* is the former or the latter. Our *Permutation* class maintains both arrays internally and supplies each on demand.

4. Design of the lower layers. While the larger part of our code deals with the design of the higher layers, most of the CPU time is actually spent in few computationally intensive loops. No advanced software paradigms are needed at this level so we concentrated on performance by carefully implementing these loops.

A major problem with C++ (also with C) is pointer aliasing, which makes code optimization more difficult for a compiler. We get around this problem by making local copies of simple variables in our kernel code. Another source of performance loss is complex numbers, since they are not a built-in in C++

```
main()
{
  /* Load the coefficient matrix and the right hand side vector. */
  SparseSymmMatrix a("a.mat");
  Vector b("b.vec");

  /* Reorder the matrix to reduce fill. */
  Permutation p(a.getSize());
  MultMinDegOrder order(a, p);
  order.run();

  /* Factor the reordered matrix. */
  SparseLwTrMatrix l(a.getSize());
  PosDefMultFrtFactor factor(a, p, l);
  factor.run();

  /* Solve triangular systems of equations. */
  Vector x(a.getSize());
  PosDefSolve solve(l, p, b, x);
  solve.run();

  /* Save the solution. */
  x.save("x.vec");
}
```

FIG. 3.3. *A direct solver for sparse, symmetric positive definite problems at the highest level*

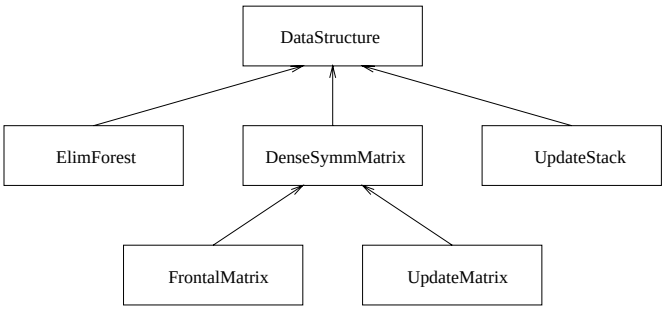

FIG. 3.4. *Structural classes used by the multifrontal numerical factorization algorithms*

data type as in Fortran. There is a template complex class in the Standard C++ library. Though this gives the compiler enough information to enforce all the rules as if it were a built-in datatype, it does not (indeed cannot) give the compiler any information about how to optimize for this class as if it were a built-in datatype.

We implemented our computationally intensive kernels both in C++ and Fortran 77. A choice between these kernels and between real and complex arithmetic can be made using compile-time switches. We defined our own class for complex numbers but we make minimal use of complex arithmetic operators, which are

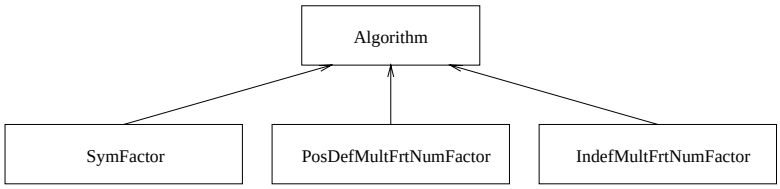

FIG. 3.5. *Some symbolic and numerical factorization algorithmic classes*

TABLE 5.1

Performance on an IBM RS/6000 for three sets of problems from fluid dynamics and acoustics. The cputimes (in seconds) and performance for the numerical factorization step are reported.

Problem	n	m	m^+	time	Mflop/s
grid9.63	3,969	15,500	104,630	0.77	34.2
grid9.127	16,129	63,756	552,871	1.70	41.6
grid9.255	65,025	258,572	2,717,313	10.89	47.4
helmholtz0	4,224	24,512	130,500	0.77	62.3
helmholtz1	16,640	98,176	639,364	4.72	77.8
helmholtz2	66,048	392,960	3,043,076	30.88	90.8
e20r0000	4,241	64,185	369,843	1.70	35.8
e30r0000	9,661	149,416	1,133,759	6.56	40.2
e40r0000	17,281	270,367	2,451,480	17.77	43.6

overloaded. The bulk of the computation is performed either in C++ kernels written in C-like style or in Fortran 77 kernels. Currently, we obtain better results with the Fortran 77 kernels.

5. Results. We report results obtained on a 66MHz IBM RS/6000 machine with 256 MB main memory, 128 KB L1 data cache and 2MB L2 cache, running AIX 4.2. Since this machine has two floating point functional units, each one capable of issuing one fused multiply-add instruction every cycle, its peak performance is theoretically 266 Mflop/s. We used the Fortran 77 kernels and we compiled the code with xlC 3.1.4 (-O3 -qarch=pwr2) and xlf 5.1 (-O4 -qarch=pwr2).

We show results for three types of problems: two-dimensional nine-point grids, Helmholtz problems, and Stokes problems, using multiple minimum degree ordering and multifrontal factorization. We use the following notation: n is the number of vertices in $G(\boldsymbol{A})$, (this is the order of the matrix), m is the number of edges in $G(\boldsymbol{A})$, and m^+ is the number of edges in $G^+(\boldsymbol{A}, \boldsymbol{P})$, the filled graph. The difference between m^+ and m represents the fill. In Table 5.1 we describe each problem using these three numbers and we also provide the cputime and the performance for the numerical factorization step, generally the most expensive step of the computation. Higher performance is obtained for the Helmholtz problems because complex arithmetic leads to better use of registers and caches than real arithmetic. We achieved performance comparable to other solvers, written completely in Fortran 77. Hence there is no performance penalty due to the object-oriented design of our solver.

We are currently implementing the solver in parallel using the message-passing paradigm. We plan to derive new classes to deal with the parallelism. Consider *FrontalMatrix* class, which stores the global indices in the *index* array and the numerical values in the *value* array. A *ParFrontalMatrix* class would need to add a *processor* array to store the owner of each column. A *ParUpdateMatrix* class may be derived in a similar way from *UpdateMatrix*. Some parallel algorithmic classes would be needed as well.

REFERENCES

[1] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object-oriented numerical software libraries*, in Modern Software Tools in Scientific Computing, Birkhauser Press, 1997.

[2] A. M. Bruaset and H. P. Langtangen, *Object-oriented design of preconditioned iterative methods in Diffpack*, ACM Trans. Math. Software (1997), pp. 50–80.

[3] A. George and J. W. H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, 1981.

[4] A. Pothen and C. Sun, *A distributed multifrontal algorithm using clique trees*, Technical Report CS-91-24, Computer Science, Penn State, Aug. 1991.

[5] C. Ashcraft, J. Lewis, and R. Grimes, *Accurate symmetric indefinite linear equation solvers*, Preprint, Boeing Information Sciences, 1995. To appear in SIAM J. Matrix Analysis and its Applications.

[6] E. Arge, A. M. Bruaset, and H. P. Langtangen, *Object-oriented numerics*, in Numerical Methods and Software Tools in Industrial Mathematics, Birkhauser, pp. 7–26, 1997.

[7] G. Booch, *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings Publishing Company, second edition, 1994.